Towards Lean 4:
An Optimized Object Model for an Interactive Theorem Prover

Sebastian Ullrich\textsuperscript{1}, Leonardo de Moura\textsuperscript{2}

\textsuperscript{1}Karlsruhe Institute of Technology, Germany \quad \textsuperscript{2}Microsoft Research, USA
The Lean theorem prover

- dependently-typed proof assistant
- small trusted kernel
- also a purely functional, eager programming language

```
inductive list (α : Type u)  
| nil : list  
| cons : α → list → list
```

```
def map (f : α → β) : list α → list β  
| []          := []  
| (x :: xs') := f x :: map xs'
```

https://leanprover.github.io
A brief history of Lean

- Lean 0.1 (2014)
- Lean 2 (2015)
  - first official release
  - fixed tactic language
- Lean 3 (2017)
  - make Lean a meta-programming language: build tactics in Lean
  - backed by a bytecode interpreter
- Lean 4 (201X)
  - make Lean a general-purpose language: native back end, FFI, ...
  - reimplement Lean in Lean
Lean 3 backend

elaborator

term

kernel

compiler

term

bytecode

interpreter

tactic execution
Towards Lean 4:
An Optimized Object Model for an Interactive Theorem Prover

Lean 4 backend

elaborator

term

kernel

term

compiler

tactic execution

IR

C++ code extraction

IR

bytecode:

interpreter

LLVM backend?
Lean 3 object model

Uniform model: every value is a tagged pointer representing one of
- a 31-bit number
- a reference to a ref-counted VM object
  - a constructor value
  - a closure
  - an arbitrary-precision integer
  - any C++ object derived from `vm_external`
Lean 3 constructor object

4 bytes reference counter
1 byte object kind
4 bytes constructor index
4 bytes #fields
4/8 bytes field #0
... ...
Lessons from Lean 3’s model

- Originally only intended for single-threaded code
  \[\Rightarrow\] no need for atomic RC!
Lessons from Lean 3’s model

- Originally only intended for single-threaded code
  ➔ no need for atomic RC!
- Eventually needed to move objects between threads
  ➔ fall back to deep-copying...
Lessons from Lean 3’s model

- Originally only intended for single-threaded code
  $\Rightarrow$ no need for atomic RC!
- Eventually needed to move objects between threads
  $\Rightarrow$ fall back to deep-copying...
- Every object is a C++ smart pointer
  $\Rightarrow$ simple to use, but no way to optimize RC ops
Lessons from Lean 3’s model

- Originally only intended for single-threaded code
  ⇒ no need for atomic RC!
- Eventually needed to move objects between threads
  ⇒ fall back to deep-copying...
- Every object is a C++ smart pointer
  ⇒ simple to use, but no way to optimize RC ops
- Core types like name and expr are not VM objects
  ⇒ need to be wrapped in vm::external for every operation
Lean 4 object model

Non-uniform model: in the lowest IR, each value has one of the types

- int8/uint8/…/uint64: unboxed primitive value
- _obj: tagged pointer to a VM object
  - a constructor, closure, or bigint
  - an array of boxed or unboxed values
  - a thunk
Lean 4 constructor object

1 byte object kind
1 byte memory kind
2 bytes constructor index
2 bytes #boxed fields
2 bytes #unboxed bytes
4/8 bytes boxed field #0
... ...
X bytes unboxed field #0
... ...

All boxed fields come first → free can still be implemented uniformly
Memory kind

- single-threaded: non-atomic RC
  - the default for heap allocations
- multi-threaded: atomic RC
  - threading primitives *upgrade* object graphs crossing threads to this kind
  - everything is immutable \(\implies\) ST object never reachable from MT object
- stack: no RC
- region: no RC
The case for ref counting

- writing a good GC is really hard

“The biggest challenge is implementing the garbage collector.”

– Multicore OCaml website

\[1\]http://ocamllabs.io/doc/multicore.html
The case for ref counting

- writing a good GC is really hard

  “The biggest challenge is implementing the garbage collector.”

  – Multicore OCaml website¹

- easier to use from other languages

¹http://ocamllabs.io/doc/multicore.html
The case for ref counting

- writing a good GC is really hard
  
  “The biggest challenge is implementing the garbage collector.”
  
  – Multicore OCaml website\(^1\)

- easier to use from other languages
- everything is immutable $\rightarrow$ no cycles!

\(^1\)http://ocamllabs.io/doc/multicore.html
The case for ref counting

- writing a good GC is **really hard**
  
  "The biggest challenge is implementing the garbage collector."
  
  – Multicore OCaml website\(^1\)

- easier to use from other languages
- everything is immutable \(\implies\) no cycles!
- explicit ref count \(\implies\) can do destructive updates on \(RC = 1\)
  - like linear types, but checked dynamically
    - dependent types are hard enough
    - more precise (but also less predictable)

\(^1\)http://ocamllabs.io/doc/multicore.html
def map (f : α → β) : list α → list β
| [] := []
| (x :: xs') := f x :: map xs'
Dynamic linearity

def map (f : α → β) : list α → list β
| [] := []
| (x :: xs') := f x :: map xs'

[compiler.llnf]
λ (f xs : _obj),
list.cases_on xs
  (let _x_1 : _obj := _dec f
   in _cnstr.0)
  (let _x_1 : _obj := _proj.0 xs,
   _x_2 : _obj := _inc _x_1,
   _x_3 : _obj := _proj.1 xs,
   _x_4 : _obj := _inc _x_3,
   _x_5 : _obj := _reset.2 xs,
   _x_6 : _obj := _apply f _x_2,
   _x_7 : _obj := list.map f _x_4
   in _reuse.1 _x_5 _x_6 _x_7)

/reset / _reuse  check for linearity at runtime
⇒ unique prefix of a list will be reused even if remainder is shared!
Dynamic linearity

Benchmarks of direct C++ implementations of

```c
list.map (+1) (list.range 4000)
```

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>Run time of map</th>
</tr>
</thead>
<tbody>
<tr>
<td>no reuse</td>
<td>214.3 µs</td>
</tr>
<tr>
<td>_reset / _reuse</td>
<td>27.7 µs</td>
</tr>
<tr>
<td>optimized reuse</td>
<td>12.3 µs</td>
</tr>
<tr>
<td>known unique</td>
<td>10.7 µs</td>
</tr>
</tbody>
</table>
Borrowing

```lean
def length : @borrowed (list α) → nat
| []        := 0
| (x :: xs') := length xs' + 1
```

```lean
[compiler.llnf]
∧ (xs : _obj),
list.cases_on xs
  0
  (let _x_1 : _obj := _proj.1 xs,
   _x_2 : _obj := length _x_1,
   in nat.add _x_2 1)
```

The `@borrowed` attribute

- delays/avoids RC operations:
  - no inc/dec when passing an argument to a borrow parameter
  - inc when returning/passing a borrowed value to a non-borrow parameter
- but prevents linear updates
Regions: minimizing startup time

Lean 3 startup does not scale well: deserializing all dependencies can take significant time and memory.
Regions: minimizing startup time

Lean 3 startup does not scale well: deserializing all dependencies can take significant time and memory

Compare with Isabelle: dependencies can be compiled into a single ML heap image
Regions: minimizing startup time

Lean 3 startup does not scale well: deserializing all dependencies can take significant time and memory

Compare with Isabelle: dependencies can be compiled into a single ML heap image
- but at most one heap can be loaded
Regions: minimizing startup time

Lean 3 startup does not scale well: deserializing all dependencies can take significant time and memory

Compare with Isabelle: dependencies can be compiled into a single ML heap image

- but at most one heap can be loaded
- still needs to be read from disk eagerly
Regions: minimizing startup time

Lean 3 startup does not scale well: deserializing all dependencies can take significant time and memory

Compare with Isabelle: dependencies can be compiled into a single ML heap image

- but at most one heap can be loaded
- still needs to be read from disk eagerly
- fragile: heaps cannot be created from an IDE session (at the moment)
Regions: minimizing startup time

Lean 3 startup does not scale well: deserializing all dependencies can take significant time and memory

Compare with Isabelle: dependencies can be compiled into a single ML heap image

- but at most one heap can be loaded
- still needs to be read from disk eagerly
- fragile: heaps cannot be created from an IDE session (at the moment)

What if we could just mmap (multiple!) regions of objects into memory?
Regions: minimizing startup time

Lean 3 startup does not scale well: deserializing all dependencies can take significant time and memory

Compare with Isabelle: dependencies can be compiled into a single ML heap image
- but at most one heap can be loaded
- still needs to be read from disk eagerly
- fragile: heaps cannot be created from an IDE session (at the moment)

What if we could just mmap (multiple!) regions of objects into memory?
- lazy loading and prefetching provided by the OS
  - proofs aren’t needed usually
Regions: minimizing startup time

Lean 3 startup does not scale well: deserializing all dependencies can take significant time and memory

Compare with Isabelle: dependencies can be compiled into a single ML heap image

- but at most one heap can be loaded
- still needs to be read from disk eagerly
- fragile: heaps cannot be created from an IDE session (at the moment)

What if we could just mmap (multiple!) regions of objects into memory?

- lazy loading and prefetching provided by the OS
  - proofs aren’t needed usually
- everything immutable
  \[\Rightarrow\] pages can even be shared by multiple Lean processes
- careful: must not touch RC
Regions: minimizing startup time

We’re investigating two approaches:

**Simple approach:** use relative pointers in region objects
- introduces branch for retrieving unboxed field
Regions: minimizing startup time

We’re investigating two approaches:

**Simple approach:** use relative pointers in region objects
- introduces branch for retrieving unboxed field

**Advanced approach:** try to `mmap` each region to its original address
- on collision: fall back to eager loading and pointer patching
- probability of a single collision between 100 dependencies of size 10 MB in 48-bit address space is ~0.018%
Regions: minimizing startup time

We’re investigating two approaches:

**Simple approach:** use relative pointers in region objects
- introduces branch for retrieving unboxed field

**Advanced approach:** try to `mmap` each region to its original address
- on collision: fall back to eager loading and pointer patching
- probability of a single collision between 100 dependencies of size 10 MB in 48-bit address space is ~0.018%

In either approach, *writing* objects to disk *does* need some transformations
Regions: minimizing startup time

For regions to work, all state to be serialized must be Lean objects
Regions: minimizing startup time

For regions to work, all state to be serialized must be Lean objects

Unboxed fields now make it feasible to reimplement core types as Lean objects!

```c
expr mk_const(name const & n, levels const & ls) {
  expr r(mk_cnstr(static_cast<unsigned>(expr_kind::Const), n, ls, expr_scalar_size(expr_kind::Const)));
  set_scalar<expr_kind::Const>(r, hash(n.hash()), hash(ls)), false, has_mvar(ls), false, has_param(ls));
  return r;
}
```
Regions: minimizing startup time

For regions to work, all state to be serialized must be Lean objects

Unboxed fields now make it feasible to reimplement core types as Lean objects!

```c++
expr mk_const(name const & n, levels const & ls) {
  expr r(mk_cnstr(static_cast<unsigned>(expr_kind::Const), n, ls, expr_scalar_size(expr_kind::Const)));
  set_scalar<expr_kind::Const>(r, hash(n.hash()), hash(ls), false, has_mvar(ls), false, has_param(ls));
  return r;
}
```

Unboxed metadata is at the end of the object

⇒ can be hidden in the Lean definition

```
inductive expr
| const : name → list level → expr
| ...
```
Implementation status

- object model runtime in C++
- core types ported to model
- optimizing compiler from Core Lean to LLNF
  - inlining, specialization, simplification
  - using join-point representation
- compiler from LLNF to old bytecode format
- model used by backends and built-ins
- writing and loading regions
- multi-threading
- borrowing
Conclusion

- A new object model customized to the needs of a theorem prover
- utilizing properties of an eager, purely functional language
- designed to avoid allocations during startup and at run time
A new object model customized to the needs of a theorem prover
utilizing properties of an eager, purely functional language
designed to avoid allocations during startup and at run time

Thank you!